A Code Generation Method For System-Level Synthesis on ASIC, FPGA and Manycore CGRA

Shuo Li, Jamshaid Malik, Shaoteng Liu and Ahmed Hemani

Royal Institute of Technology
Forum, Isafjördsgatan 39
SE-164 40, Stockholm, Sweden
{shuol, jamshaid, liu2, hemani@kth.se}

ABSTRACT

This paper presents a code generation method that translates an intermediate Register-Transfer Level (RTL) model of a system into its corresponding VHDL code for ASIC and FPGAs and MATLAB functions for manycores CGRAs. The intermediate representation consists of Function Implementation (FIMPs) and the glue logic. FIMPs are VHDL design units for the ASIC and FPGA implementation styles and MATLAB function templates for the CGRA implementation style, while the glue logic is a compact data structure storing Global Interconnect and Control (GLIC) information.

The automatically generated implementation codes increase the resource usage by 1.5% on the average while reducing total design effort by two orders of magnitudes.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic Synthesis, Hardware Description Languages*

General Terms

Design

Keywords

Code Generation, System-Level Synthesis, Function Implementation, Global Interconnect and Control

1. INTRODUCTION

Code generation, in computer science, is the process that converts the intermediate representation of source code into a form that can be executed by a machine [1]. In digital design flow, code generation produces suitable descriptions at the target abstraction level. For example, code generation at logic synthesis level produces discrete netlists of logic gates, while code generation in High-Level Synthesis (HLS) produces Register-Transfer Level (RTL) descriptions, which is normally modeled by VHDL, Verilog or SystemC codes [2].

MES '13 Tel-Aviv, Israel

In this paper, we present a code generation method used in the System-Level Architectural Synthesis (SYLVA) framework [3]. By using this method, a suitable description will be generated from the intermediate representation used in SYLVA. This description can be a VHDL module for ASIC and FPGA implementation style, or a composite MATLAB function for Coarse-Grained Reconfigurable Architecture (CGRA), which is widely used in the DSP community [4].

This paper is organized as follows:

Section 2 gives a brief introduction to the SLYVA framework, FIMP and GLIC. Section 3 describes the proposed code generation method with sufficient details. Section 4 lists the experimental result and Section 5 gives the conclusion and the future work.

2. PRELIMINARIES

In this section, we will give a brief introduction to the SYLVA framework and its basic building blocks. These include Function Implementations (FIMPs) [5] and global interconnect and control.

2.1 System-Level Architectural Synthesis

The SYLVA framework translates a DSP system into a corresponding RTL description (VHDL design units for ASIC/FPGA) or a MATLAB function for CGRA; the MATLAB code is translated into configware (software) by CGRA specific compiler.

2.1.1 SYLVA Overview

The SYLVA flow refines an SDF graph into a representation that can be synthesized into a manufacturable design by existing commercial synthesis tools for ASIC and FPGA implementation styles and compilers for CGRAs.

The salient feature of SYLVA is that it performs system level synthesis in terms of pre-characterized FIMPs - Function Implementations as reusable building blocks. Here function refers to DSP functions like FFT, Viterbi, FIR etc. Synchronous Data Flow (SDF) [6] actors represent functions. Pre-characterization implies that these functions are implemented down to layout level and the area, energy and latency costs are characterized taking into account the sign-off quality post layout design data. In contrast to SYLVA, HLS tools synthesize in terms of pre-characterized micro-architectural units like adders, multipliers, registers, multiplexers etc. Use of FIMP enables SYLVA to deal with more complex system level specifications, reduce the search space by using coarser granularity building blocks (FIMPs vs. micro-architectural blocks) to reach more optimal solutions and the use of pre-characterized coarse grain FIMPs brings predictability and improves reuse of design.

SYLVA targets DSP sub-systems like modems and codecs. It

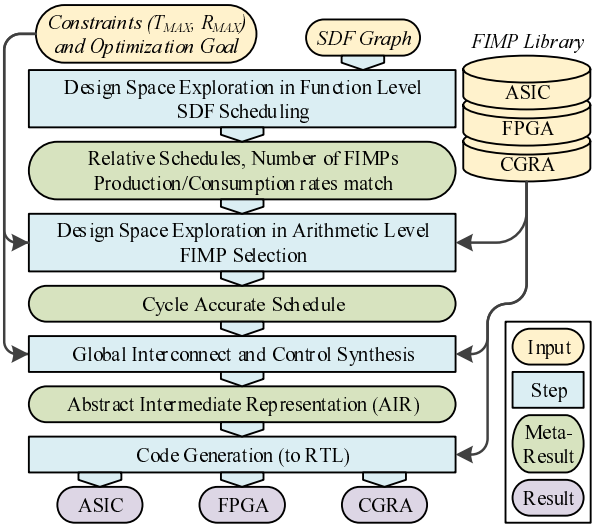

Figure 1: SYLVA Flow for ASIC, FPGA or CGRA

accepts two timing constraints: total latency (T_{MAX}) and sampling rate (R_{MAX}). During the synthesis process, SYLVA accesses FIMP libraries. These libraries contain pre-characterized implementation of DSP functions. Each function is available in multiple implementations differing in dimension and area, energy and latency costs, e.g. FFT implementations can be available with single or multiple butterflies. FIMP libraries are available for three implementation styles: ASIC (Standard Cells), FPGA and CGRA, and in different technology nodes in each style and potentially in different technology nodes (ASIC, CGRA and FPGA families).

As shown in Figure 1, the SYLVA flow has three major steps: design space exploration (including SDF scheduling and FIMP selection as shown in the top two blue blocks in Figure 1), global interconnect and control synthesis and code generation. FIMPs have three views in the FIMP libraries corresponding to the three steps of the SYLVA flow. The first view is a cost model that is used during the design space exploration phase The second view is the interface model that is used during the global interconnect and storage synthesis phase, and the third view is the implementation model that is used during the code generation phase. FIMP is elaborated with more details in Section 2.2.

The focus of this paper is the code generation step. However, to explain this step, it is essential to briefly describe the two preceding steps.

SYLVA Design Space Exploration (DSE) is carried out at two levels. At the first level, the design space is explored in terms of number of FIMPs and the second level in terms of type of FIMPs. This is similar to HLS tools exploring design space in terms of number of arithmetic units and type of arithmetic units. The end result of the DSE step is that SYLVA knows the number and type of FIMPs required that meets the T_{MAX} and R_{MAX} constraints and minimizes a cost function in terms of energy and area. An implication of DSE is the synthesis of a cycle accurate schedule that specifies when a FIMP is fired initially and the delay between its repetitive firing. Note that, SDF enables us to create a static schedule. For example, assume we have an input SDF graph as shown in Figure 2(a). It describes part of the 802.11a physical layer. Assume that the latency for all the functions that implement the actors are all one clock cycle. After the two level DSE, we get the scheduled SDF graph shown in Figure 2(b) and the cycle accurate schedule

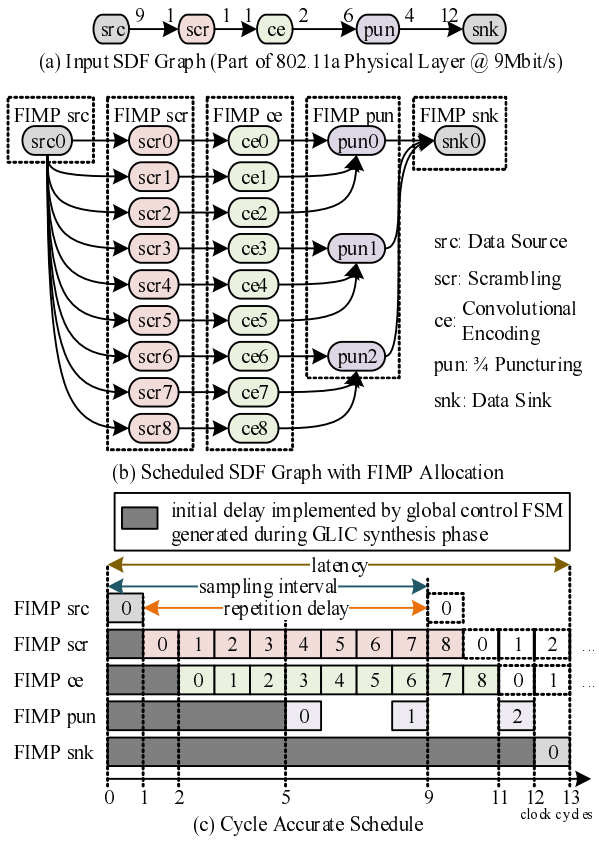

Figure 2: SYLVA Example, Part of 802.11a Physical Layer

shown in Figure 2(c). The numbers are the indexes of the FIMP executions.

The next step of SYLVA is to generate the Global Interconnect and Control (GLIC) to transfer data between the FIMPs and orchestrate their executions while fulfilling the cycle accurate schedule created in the DSE step. The end result of this GLIC synthesis step is an Abstract Intermediate Representation (AIR). AIR is implementation style (ASIC, FPGA, CGRA) agnostic and it is a netlist of AIR Building Blocks (ABBs).

Each ABB, shown in Figure 3, is a standardized template that encapsulates a FIMP with infrastructural elements to implement the global interconnect and control. The infrastructural elements are input and output selectors, and FIMP and buffer controllers. In essence, ABBs are wrappers around FIMPs that help implement the global interconnect and control. For each FIMP that has been selected in the design space exploration step, SYLVA customizes an ABB to implement the connectivity to other FIMPs by customizing the input and output selectors and controlling the timing of the FIMP execution and input and output process to fulfill the cycle accurate schedule by customizing the FIMP and the buffer controllers.

Additionally, SYLVA generates a global control FSM to trigger the ABBs, the first time to implement the initial delay (see Figure 2 showing the cycle accurate schedule). After the initial triggering by the Global FSM (initial delay shown as the dark blocks in Figure 2(c)), the local ABB FIMP control FSMs trigger their respective FIMPs repeatedly after a delay (repetition delay in Figure 2(c)) according to the cycle accurate schedule. The buffer control FSMs also trigger the reading and writing to right buffers at right

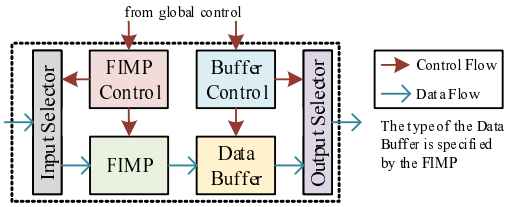

Figure 3: AIR Building Block Structure

time in accordance with the cycle accurate schedule. This detail is not shown in Figure 2 as it would clutter the diagram a lot.

Once an AIR (netlist of ABBs + Global FSM) has been generated for a design represented by an SDF graph, SYLVA generates the code that is necessary for the implementation that is specific to the implementation style and the selected FIMP library. The focus of this paper is how this code generation step works for the three implementation styles: ASIC, FPGA and CGRA.

For the ASIC and FPGA implementation styles, the FIMP library has an RTL version of the FIMP and the ABB template/parametric design that are configured and generated according to the specifications in the AIR. The generated RTL code is in VHDL and synthesized by commercial synthesis tools.

For the CGRA, we have used Dynamically Reconfigurable Resource Array (DRRA) [7] as an example along with its compiler VESYLA [8]. The templates for the FIMP and the infrastructural components of ABB are all modeled as MATLAB functions. These MATLAB functions are then translated into configware for DRRA by its compiler VESYLA; which translates each individual functions into an executable code for DRRA. DRRA is a grid like structure of DRRA cells [9] interconnected by a nearest neighborhood crossbar like interconnect scheme. The nearest neighborhood concept allows each DRRA cell to reach within a window of n cells; in the present DRRA fabric n = 3. The input, output selectors of ABB are realized by programming the interconnect fabric. The FIMP and buffer control FSMs are implemented by code in the sequencers and the AGUs in register files and memory banks. Note that each DRRA cell has its own sequencer and register file. And each register file has two read and two write ports, each port with its own programmable AGU that allows programming in both spatial and temporal sense [10]. An example of programming in spatial sense is reading/writing from address a to b in increments of c. And an example of programming in temporal sense is to insert arbitrary delays between each reading/writing and an initial and an end delay. DRRA also has a hierarchical control scheme [11] that allows one DRRA cell to control other DRRA cells, like in a hierarchical FSMs. This feature of DRRA is used to implement the global FSM.

2.2 Function Implementation (FIMP)

FIMPs are used to model the implementation of DSP functions. FIMPs have three views in the FIMP libraries corresponding to the three steps of the SYLVA flow. The first view is a *Cost Model* that is used during the design space exploration phase. The second view is the *Interface Model* that is used during the global interconnect and storage synthesis phase, and the third view is the *Implementation Model* that is used during the code generation phase.

2.2.1 Cost Model

In SYLVA, one function may have multiple FIMPs with the same name and numbers of input and output tokens but with the different degrees of parallelism. For example, assume we have a 64-point FFT with the name "FFT" and 64 complex numbers as input and output. This can be implemented with be multiple FIMPs that have one or multiple butterfly elements. They all have the same name and same I/O data structure but different degrees of parallelism resulting in different execution cost. To select the optimal FIMPs for the final system implementation [5], the execution costs of all FIMPs for all used functions are examined. This includes resource usage, average energy consumption for one execution and execution latency.

The resource usage is in terms of gate count for ASIC, logic element, memory element and other resource count for FPGA, and processing and storage element count for CGRA.

2.2.2 Interface Model

This model describes the I/O information of the FIMP. It has the number of I/O data tokens, number of I/O ports and number of data tokens on each I/O port. Currently, we assume the data tokens are linearly distributed. For example, data token a, b, c, d on two input ports are distributed as a and b on the first port, and c and d on the second port.

The system designer specifies the functions by providing the function names and the expected I/O information in the system SDF graph. They are then used in the function level design space exploration for selecting the suitable number of FIMPs for all functions (actors) in the SDF graph.

2.2.3 Implementation Model

The implementation model is a code template for generating the final implementation code of a FIMP by using automated code generation techniques such as [12] and [13]. It is a parametric VHDL design unit for ASIC/FPGA, or a MATLAB function for CGRA. Note that the VHDL code for FPGA may also includes data storage part of a function and some FPGA-specific resource mapping, e.g. DSP, memory element, etc. The method to generate the actual implementation code based on the template will be discussed in Section 3.2. Note that the implementation of the output buffer of the FIMP is also defined in this model.

For ASIC and FPGA implementation styles, a FIMP is fired by setting control signal in the VHDL module. For CGRA implementation style, a FIMP is fired by invoking the corresponding function by changing the program counter to the right value at right time.

2.3 Global Interconnect and Control (GLIC)

GLIC represents the execution and communication of all the DSP functions. It is stored in a data structure that each FIMP with its execution and communication schedule is combined as an AIR Building Block (ABB). Besides the FIMP, each ABB has five components, which are *data buffer*, *input selector*, *output selector*, *FIMP control* and *buffer control*. In the AIR, there is also a global control for initialization purpose.

The ***Data Buffer*** stores and sends out FIMP output. It is also considered as a part of GLIC since an output buffer for one FIMP is always an input buffer for another FIMP. And we can assign extra buffers to increase the overall system throughput [14].

The ***Input Selector*** is a multiple input/output switch that connects the data sources to the input ports of the FIMP based on the communication schedule. Similarly, the ***Output Selector*** is for establishing physical interconnection between the data buffer and its data sink(s).

The ***FIMP Control*** is an FSM. It controls the FIMP according to the execution schedule. It also commands the input selector to establish data communication channel based on the communication schedule. The ***Buffer Control*** is also an FSM. It controls the data buffer for *a)* storing the output of FIMP in the same ABB and

b) outputting data to the data sink FIMP in another ABB. Buffer control also commands the output selector to establish data communication channel based on the communication schedule.

The clock cycle values in the execution and communication schedules refer to two different time point. For the execution schedule, the referred time point is the first time a FIMP is executed in a system iteration. For the communication schedule, it is the first time a data buffer takes the output data from its FIMP in a system iteration. The process to start the FIMPs and data buffers at their corresponding time is called initialization and it is implemented by the ***Global Control*** (an FSM).

3. CODE GENERATION

The code generation in the SYLVA framework converts AIR into implementation codes (VHDL module for ASIC/FPGA, MATLAB function for CGRA). Figure 4 shows an example AIR for the example SDF shown in Figure 2(a). Each arrow is a data transfer for one actor in the scheduled SDF graph in Figure 2(b). For example, the ABB scr has nine scr actors executed on it. Each scr actor has two arrows that represent its input and output.

3.1 Code Generation Overview

The proposed code generation procedure is illustrated in Figure 5. Each dashed block (except the left and the right most ones) represents one complete code generation procedure for one FIMP. The interconnection edges represent the procedural dependencies. The code generation for FIMPs and data buffers do not depend on any other component. However, the code generation for an output selector has to wait for the code generation for its data buffer(s) since one data buffer may be mapped to a single memory bank and while other data buffer may be mapped to multiple memory banks. The situation could also be the other way around, i.e., multiple data buffers could be realized by a single memory bank as well. The output selector needs to know the relationship between the data signals represented in the ABB and the actual implementation, which can only be known after the code generation of the data buffer is done. Similarly, the code generation of the buffer control can only start after the code generation of the output selector. After the code generation of the FIMP and output selector, the code generation of the input selector starts. The code generation of the FIMP control can start only after the code generation of the input selector.

After the code generation for all the components, the code for the global control is generated and combined with other components.

3.2 Generate FIMP

The actual implementation codes are generated based on the implementation templates stored in the FIMP library. For ASIC and FPGA, these are in the form of parametric VHDL design units with generic parameters. Multiple FIMPs may share the same VHDL entity but each FIMP instance has its own values of the generic parameters. For example, consider a 16-tap FIR symmetric filter with 8-bit input and output data. The implication is that we need 8 MACs for maximum parallelism. Assuming that it can be implemented by four FIMPs with different number of multiply-accumulators (1, 2, 4, or 8). All the four FIMPs share the same VHDL entity as the implementation template called *FIR16_1_1*. The first *_1* means its input data size is one and the other *_1* means its output data size is also one. The code segment containing port and generic sections is shown below. By removing the generic section from the VHDL code and replacing *stage* with 1, 2, 4 or 8, we can generate the VHDL design units for four different FIMPs. If there are two FIMPs for 16-tap FIR filters with the same FIMP type (16FIR_1_1), they will have different FIMP names after instantiation (*16FIR_1_1_FIMP_0* and *16FIR_1_1_FIMP_1*).

```
1  entity FIR16_1_1 is
2  generic (
3     stage : integer range 1 to 8 := 1;
4     coeff : in coeff (7 downto 0) :=...
5  );
6  port (
7     clock : in std_logic;
8     nreset : in std_logic;
9     din : in integer range -128 to 127;
10    dout : out integer range -128 to 127
11 );
12 end FIR16_1_1;
```

For CGRA, the code generation is similar to the one for ASIC and FPGA as explained above. For the 16-tap FIR symmetric filter example, a constant *MACs* is used for the number of iterations in the outer loop reflecting the number of stages. The code segment is shown below.

```
1  % Function:  FIR16_1_1
2  for j = 1:MACs % SIMD threads:  MACs
3     for i = 1:(8/MACs)
4     % Inner loop multiplications and additions
```

Line 1 provides the architecture of the FIMP. Line 2 determines the number of MACs. The comment is a pragma that marks the number of intended SIMD threads. This comment is not executed by MATLAB but provides information to the the CGRA compiler. It only shows the method to provide information to the CGRA compiler and the actual acceptable pragma may not in this form. Line 3 determines the number of iterations. By replacing *MACs* with 1, 2, 4 or 8, we can generate the MATLAB function files for the four different FIMPs. Although there is no difference when these MATLAB functions are executed in MATLAB environment, they provide information to the CGRA compiler to generate FIR filter with the right degree of parallelism. The pseudo-code for generating a FIMP is shown as follows.

```
1  CFG = Configuration of the implementation template;
2  Temp = Implementation template;
3  if ASIC or FPGA do
4     Remove generic section in Temp;
5  end
6  Replace parameters in Temp with their values in CFG;
7  if multiple instances in system do
8     Change FIMP name;
9  end
10 result = Temp;
```

In line 1, *CFG* represents the names and the values of the generic parameters (for the VHDL entity) or constants (for MATLAB function) to be used in implementation template. For example, if the number of stages is two, then *CFG* = {stage, 2} for ASIC/FPGA, and *CFG* = {REPLACE_stage, 2} for CGRA. In line 2, *Temp* represents the implementation template, which is a VHDL entity with generic section for ASIC/FPGA or a MATLAB function containing constants without value assignment. In line 7, *result* is the actual implementation code for the FIMP. Since there is no generic section in the MATLAB file, it is no need to be removed.

3.3 Generate Data Buffer

In ASIC implementation style, the data buffers are instantiated as on-chip SRAM blocks. Note that there can be multiple data buffer design units in one ABB. The type of the data buffer used for a FIMP is specified by the FIMP, i.e. the data buffer implementation is part of the FIMP implementation as described in Section 2.2.3

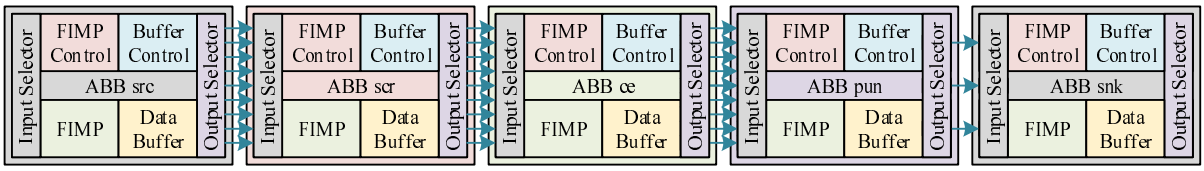

Figure 4: AIR Example

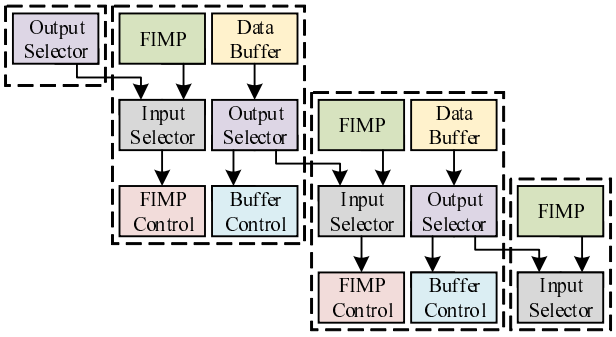

Figure 5: Code Generation Steps

Each data buffer in the ABB is instantiated as one or more memory blocks. The number of memory blocks is determined by the output port of the FIMP.

For example, each data buffer is instantiated as one memory block for the example 16-tap FIR filter, since it has only one output port. For an RGB to YUV converter, which has three separate output ports for Y, U and V signals, each data buffer is instantiated as three memory blocks. Following code segment is part of the VHDL module for one memory block.

```
1  entity SYLVADataBuffer is
2  generic (
3     capacity : integer := 256;
4     data_width : integer := 8
5  );
6  port (
7     clock : in std_logic;
8     nreset : in std_logic;
9     enable : in std_logic;
10    nwrite : in std_logic;
11    address : in integer;
12    din : in std_logic_vector
        (data_width-1 downto 0);
13    dout : out std_logic_vector
        (data_width-1 downto 0)
14 );
15 end SYLVADataBuffer;
```

The pseudo-code for generating the actual implementation code for ASIC implementation style is shown as follows:

```
1  result = ∅;
2  i = 0, j = 0;
3  foreach ABB do
4     foreach output port P in the FIMP do
5        foreach Data buffer D in the ABB do
6           Temp = entity SYLVADataBuffer;
7           Replace capacity in Temp with the capacity of D;
8           Replace data_width in Temp with the data width of P;
9           Append _j_i after SYLVADataBuffer in Temp;
10          i = i + 1;
11          Add Temp into result;
12       end
13    end
14    j = j + j;
15 end
```

In the FPGA implementation style, the data buffers can either be implemented in ASIC style as shown above, or by using existing block memory. When using the FPGA specific memory resources, those memory resources have to be pre-wrapped to provide the same interface as the ASIC memory blocks. This resource mapping is decided during the FIMP selection step in SYLVA (Section 2.1). The following pseudo-code shows generating implementation code with specified memory blocks:

```
1  result = ∅;
2  i = 0;
3  foreach memory block M do
4     Temp = Wrapped M;
5     // Providing the same interface as ASIC memory blocks;
6     Add Temp into result;
7  end
```

In the CGRA implementation style, the implementation method is the same as in ASIC. However, the actual implementation is not the VHDL module but a vector with *capacity* number of elements. The type of the elements in one vector is decided by the output data type of the FIMP. For example, the output of a RGB to YUV converter has three components (Y, U and V) and each component is a decimal number. Thus, the data buffer for this RGB to YUV converter is instantiated as three vectors, whose elements are decimal numbers. If the converter is for one RBG to one YUV conversion, each vector has one element. This is shown in the following pseudo-code:

```
1  result = ∅;
2  i = 0;
3  foreach output port P in the FIMP do
4     foreach data buffer D in the ABB do
5        <name> = FIMP name concatenated with M_i;
6        i = i + 1;
7        <d> = capacity of D;
8        Temp = "<name> = zeros(<d>)";
9        Add Temp into result;
10    end
11 end
```

3.4 Generate Output Selector

The output selector is a switch that connects the input ports to the corresponding output ports according to the control signals from the buffer control. The number of input ports of the output selector equals to the number of data buffers. The number of output ports equals to the number of data sink actors in the scheduled SDF graph multiplied by the number of output ports of the FIMP.

For example, consider a 16-tap symmetric FIR filter, which is feeding an FFT and an 8-tap FIR filter. In this case, the output selector of the 16-tap FIR has one input port and two output ports.

In ASIC/FPGA implementation styles, the interface of the VHDL entity of an output selector is generated from the template shown as follows: (FN is short for function name)

```
1  entity SYLVAOutputSelector<FN> is
2  port (
3    clock : in std_logic;
4    osel : in OS<FN>);
5    din : in inT<FN>);
6    dout : out outT<FN>)
7  );
8  end SYLVAOutputSelector<FN>;
```

During code generation, <*FN*> will be replaced by the FIMP name. For example, if there are two FIMPs for 16-tap FIR filters with the same FIMP type (16FIR_1_1), they will have different FIMP names (*16FIR_1_1_FIMP_0* and *16FIR_1_1_FIMP_1*). Line 4 declares the selection signal *osel*, which in type *OT*<*FN*>. This type is a vector. Each element in this vector is an integer from 0 to the number of output ports. Zero means no connection and other values are the one-based index of the output ports. The value of the *Nth* element indicates which input port is connected with the *Nth* output port.

For example, when *osel* = {0, 4, 1, 0, 0, 0}, the second input port is connected to the fourth output port, the third input port is connected to first output port and other output ports are set to high impedance state. The output selector is implemented via a set of case statements. For the sake of brevity, the actual code is being skipped.

Lines 5 and 6 declare the input/output ports. For *din*, the number of elements equals to the number of input ports.

The type of each element is the same as the corresponding data buffer output port. The following pseudo-code show how to build the VHDL types and store it in the type library for this system.

```
1  TL = Type library file in VHDL format;
2  <I> = # data buffers * # FIMP output ports;
3  <O> = # sink actors * # FIMP output ports;
4  // for port: osel;
5  Add "type OS<FN> is array (<O>-1 downto 0) of integer;" into TL;
6  // for port: din;
7  Temp = "type IE<FN> is (";
8  foreach data port P in FIMP output ports do
9     Concatenate Temp with type of P;
10 end
11 Concatenate Temp with ");";
12 Add Temp into TL;
13 Add "type inT<FN> is array (<I> downto 0) of IE<FN>;" into TL;
14 // for port: dout;
15 Add "type outT<FN> is array (<O> downto 0) of IE<FN>;" into TL;
16 Replace "<FN>" in TL with the FIMP name;
17 result = entity SYLVAOutputSelector;
18 Replace "<FN>" in result with the FIMP name;
```

In CGRA implementation style, the output selector is constructed via a set of switch statements. Again, for the sake of brevity, the actual code is being skipped. The pseudo-code for generating codes of an output selector for CGRA is shown as follows:

```
1  result = MATLAB function SYLVAOutputSelector;
2  i = 0;
3  foreach data sink do
4     Temp = "switch In_" concatenate with i;
5     i = i + 1;
6     foreach data buffer D do
7        Add "case " concatenate with j into Temp;
8        Add "dout_" concatenate with i into Temp;
9        Add "= din_" concatenate with j into Temp;
10       Add "otherwise \n end" into Temp;
11       j = j + 1;
12    end
13 end
```

3.5 Generate Input Selector

The code generation for the input selector is similar to the code generation of the output selector. The number of output ports, which connect the input selector to the FIMP, equals to the number of FIMP input ports. Similarly, the number of input ports, which connect the data sources to the input selector, equals to the number of data sources multiplied by the number of FIMP input ports. The generation procedure is exactly the same.

3.6 Generate Buffer/FIMP/Global Control

A buffer control is an FSM controlling the data buffers and the output selector. For ASIC/FPGA, we use the generation method described in [15] for FSM code generation. An interested reader may refer to [15] for the details. For CGRA, one FSM is translated into two separate functions, which are *state control function* and *state output function*. Like the buffer control, the FIMP control and the global control are also FSMs. The code generation method is exactly the same as the buffer control.

3.7 Combine All Components

After we have the implementation codes for all the individual components (FIMP, data buffer, output selector, input selector, buffer control and FIMP control), we could combine them together in one VHDL module or MATLAB function. The combination algorithm is shown in Algorithm 1.

4. EXPERIMENTAL RESULT

In this section, we evaluate the code generation method for ASIC, FPGA and CGRA by applying SYLVA framework on five examples. The five example systems are a) FFT connected with two FIR filters, b) the correlation algorithm in Rake receiver, c) sigma-delta demodulator, d) JPEG encoder and e) part of a MPEG2 encoder (SDF cannot model varying data rate). The SDFs of them are shown in Figure 6.

To synthesis the generated codes on ASIC, FPGA and CGRA, we used Cadence RTL Encounter, Altera Quartus II and VESYLA respectively. VESYLA [8] is the compiler for Dynamically Reconfigurable Resource Array (DRRA) [7]. For each implementation style, we list the total resource usage and the Global Interconnect and Control (GLIC) resource usage achieved by manual design and SYLVA. For ASIC/CGRA, the resources are in terms of logic gates. For FPGA, the resources are in terms of logic elements.

First, the interconnect and control logic was generated manually. On average, a manual design requires approximately two days. The code for these examples were generated by SYLVA automatically in less than 10 minutes on average. This translates to a decrease in the total design effort by an order of two magnitudes. Both design methods (manual and SYLVA) share the same type of FIMPs and same amount of buffers. In Tables 1, 2 and 3, we compare the manually generated codes with the automatically generated codes in the SYLVA framework. The data buffers are considered as part of FIMP thus not included in the GLIC resource usage.

4.1 ASIC Implementation Style

For the ASIC implementation style, TSMC 65nm was used as the target technology node. The synthesis result is shown in Table 1. Note that the MPGE2 encoder contains only the parts around

```
if ASIC or FPGA then
    Top = final implementation VHDL entity;
    Add all components into architecture body;
    foreach component do
        Add its port into architecture body of Top;
        // except the clock and reset signals;
    end
    foreach ABB B do
        Connect the components of B in Top;
        Connect the output selector with its data sink ABB in
        Top;
    end
    Connect the ABB of the source actor in the scheduled
    SDF;
    Connect the ABB of the sink actor in the scheduled SDF;
    Include the library files generated for input/output selector;
else
    Top = final implementation MATLAB function;
    Add all functions for all the components;
    foreach component do
        Add its inputs and outputs as variables into Top;
    end
    foreach ABB B do
        Connect the components via the variables in Top;
        Connect the output selector with its data sink ABB in
        Top;
    end
    Connect the ABB of the source actor in the scheduled
    SDF;
    Connect the ABB of the sink actor in the scheduled SDF;
end
```

Algorithm 1: Combining All Components

the motion estimation for 720x480@25fps. The JPEG encoder is for 1920x1080 image and have a given timing constraint of 30 fps. The automatically generated GLIC takes a maximum of 4.1 *k* gates (5.7% of the total area usage).

4.2 FPGA Implementation Style

For the FPGA implementation style, we used Altera Cyclone III that also uses TSMC 65nm technology node. The synthesis result is shown in Table 2. The automatically generated GLIC takes a maximum of 405 logic elements (11.5% of the total Logic Elements (LEs)).

4.3 CGRA Implementation Style

For the CGRA implementation style, we use DRRA. It is a CGRA with cross-bar NoC communication scheme and its compiler VESYLA [8] supports MATLAB function as input system model. The target technology node for DRRA is also TSMC 65nm. The synthesis result is shown in Table 3. DRRA supports a resource usage optimization method called CRASIC [16]. It shrinks the resource usage by cutting-off unused logics in the DRRA fabric. The numbers shown in Table 3 are the numbers after CRASIC. The automatically generated GLIC takes a maximum of 6.0 *k* gates (1.2% of the total area usage). The DRRA implementation is comparable with the ASIC one since it takes 2x to 5x more gates than ASIC and provides reconfigurability.

5. CONCLUSION AND FUTURE WORK

The proposed code generation method used in the SYLVA frame-

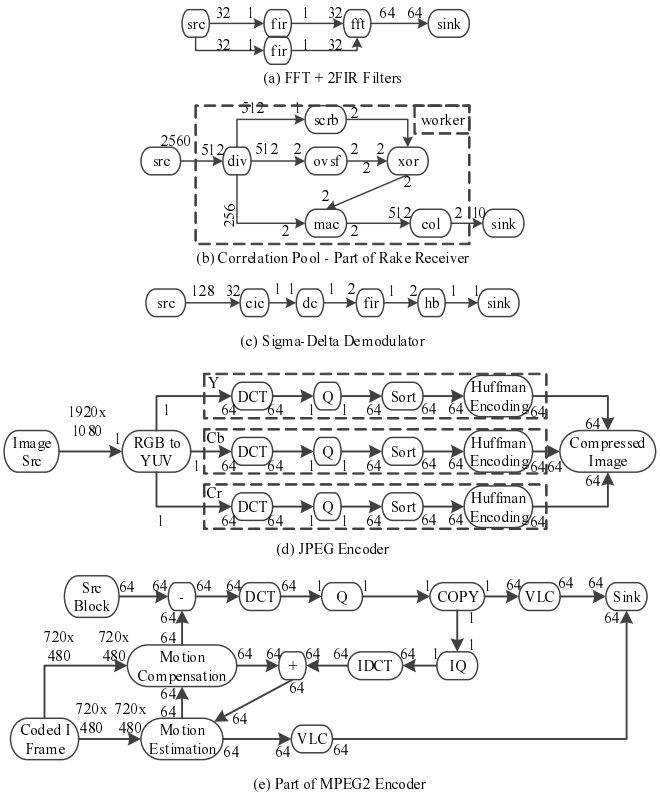

Figure 6: Example Systems

Example	Method	Area	GLIC	
		*k*gates	*k*gates	%
FFT+2FIR	Manual	76.0	2.0	2.6
	SYLVA	76.4	2.4	3.1
Correlation Pool	Manual	68.0	2.0	2.9
	SYLVA	70.0	4.0	5.7
Demodulator	Manual	51.0	0.3	0.6
	SYLVA	53.0	2.3	4.3
JPEG Encoder	Manual	365.9	1.8	0.5
	SYLVA	366.0	2.2	0.6
MPEG2 Encoder	Manual	452.0	1.8	0.4
	SYLVA	454.2	4.1	0.9

Table 1: ASIC Synthesis Result

Example	Method	Resource	GLIC	
		LEs	LEs	%
FFT+2FIR	Manual	1225	20	1.6
	SYLVA	1250	45	3.6
Correlation Pool	Manual	3.5*k*	402	11.5
	SYLVA	3.5*k*	402	11.5
Demodulator	Manual	2156	99	4.6
	SYLVA	2176	120	5.5
JPEG Encoder	Manual	16.0*k*	208	1.3
	SYLVA	16.2*k*	405	2.5
MPEG2 Encoder	Manual	7092	43	0.6
	SYLVA	7152	100	1.4

Table 2: FPGA Synthesis Result

Example	Method	Area	GLIC	
		*k*gates	*k*gates	%
FFT+2FIR	Manual	362	2.4	0.7
	SYLVA	363	2.6	0.7
Correlation Pool	Manual	403	2.7	0.7
	SYLVA	405	5.0	1.2
Demodulator	Manual	280	0.4	0.1
	SYLVA	283	2.9	1.0
JPEG Encoder	Manual	923	2.5	0.3
	SYLVA	923	2.6	0.3
MPEG2 Encoder	Manual	843	2.5	0.3
	SYLVA	846	6.0	0.7

Table 3: CGRA Synthesis Result

work generates VHDL module (for ASIC/FPGA) or composite MATLAB function (for CGRA) implementations from the Abstract Intermediate Representation (AIR). The code generation process is automatic and efficient. This code generation can also be used in standalone mode. In the near future, we plan to spend more research effort on the code optimization to achieve more compact implementation codes. For example, the size of the type library used for the ASIC/FPGA VHDL modules can be reduced by merging equivalent types.

6. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

[2] P. R. Panda, "SystemC - A Modeling Platform Supporting Multiple Design Abstractions," in *The 14th International Symposium on System Synthesis*, 2001, pp. 75–80.

[3] S. Li, N. Farahini, and A. Hemani, "Global Control and Storage Synthesis for a System Level Synthesis Approach," in *The 21st Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[4] C. Moler. (2006, Jan.) The Growth of MATLAB and The MathWorks over Two Decades. [Online]. Available: http://www.mathworks.com

[5] J. Fahimeh, S. Li, and A. Hemani, "Optimal Selection of Function Implementation in a Hierarchical Configware Synthesis Method for a Coarse Grain Reconfigurable Architecture," in *the 14th Euromicro Conference on Digital System Design (DSD)*, 2011.

[6] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, Sept. 1987.

[7] M. A. Shami, "Dynamically Reconfigurable Resource Array," Ph.D. dissertation, KTH, 2012.

[8] O. Malik, A. Hemani, and M. A. Shami, "A Library Development Framework for a Coarse Grain Reconfigurable Architecture," in *the 24th International Conference on VLSI Design*, 2011.

[9] M. A. Shami and A. Hemani, "Partially Reconfigurable Interconnection Network for Dynamically Reprogrammable Resource Array," in *The IEEE 8th International Conference on ASIC*. IEEE, 2009, pp. 122–125.

[10] M. A. Shami and A. Hemani, "Address Generation Scheme for a Coarse Grain Reconfigurable Architecture," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2011, pp. 17–24.

[11] M. A. Shami and A. Hemani, "Control Scheme for a CGRA," in *The 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2010, pp. 17–24.

[12] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC - Automatic Code Generation for VHDL," *International Journal of Reconfigurable Computing*, 2009.

[13] S. Li, G. Chen, and A. Hemani, "A Code Reuse Method for Many-Core Coarse-Grained Reconfigurable Architecture Function Library Development," in *The 13th International Symposium on Integrated Circuits (ISIC)*. IEEE, 2011, pp. 512–515.

[14] Y. Joo and N. McKeown, "Doubling Memory Bandwidth for Network Buffers," in *the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1998.

[15] A. T. Abdel-Hamid, M. Zaki, and S. Tahar, "A Tool Converting Finite State Machine to VHDL," in *Canadian Conference on Electrical and Computer Engineering*, vol. 4, 2004, pp. 1907–1910 Vol.4.

[16] CRASIC: Customisation of Coarse Grain Reconfigurable Architectures. https://drra.wikispaces.com/CRASIC+Project.